浙江大学

本科实验报告

课程名称：操作系统

姓　　名：

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：

指导教师：李环

2024 年 10 月 12 日

浙江大学操作系统实验报告

实验名称：RV64 内核线程调度

电子邮件地址：____手机：_____

实验地点：曹西 503 实验日期：2024 年 10 月 12 日

一、实验目的和要求

了解线程概念，并学习线程相关结构体，并实现线程的初始化功能

了解如何使用时钟中断来实现线程的调度

了解线程切换原理，并实现线程的切换

掌握简单的线程调度算法，并完成简单调度算法的实现

二、实验过程

（一）准备工程

从本仓库 src/lab2 同步以下代码：

```
.
├── arch
│   └── riscv
│       ├── include
│       │   ├── mm.h
│       │   └── proc.h
│       └── kernel
│           ├── mm.c        # 一个简单的物理内存管理接口
│           └── proc.c      # 本次实验的重点部分，进行线程的管理
├── include
│   ├── stdlib.h           # rand 及 srand 在这里（与 C 语言 stdlib.h 一致）
│   └── string.h            # memset 在这里（与 C 语言 string.h 一致）
└── lib
    ├── rand.c              # rand 和 srand 的实现（参考 musl libc）
```

```
    └── string.c            # memset 的实现
```

本实验中中我们需要一些物理内存管理的接口，调用 mm.c 中的 kalloc 即可申请一个 4KiB 的物理页；

由于引入了简单的物理内存管理，需要在 _start 的适当位置调用 mm_init 函数来初始化内存管理系统，代码如下：

```
1.    la sp, boot_stack_top #store the address of stack top
2.    jal mm_init
3.    jal task_init
```

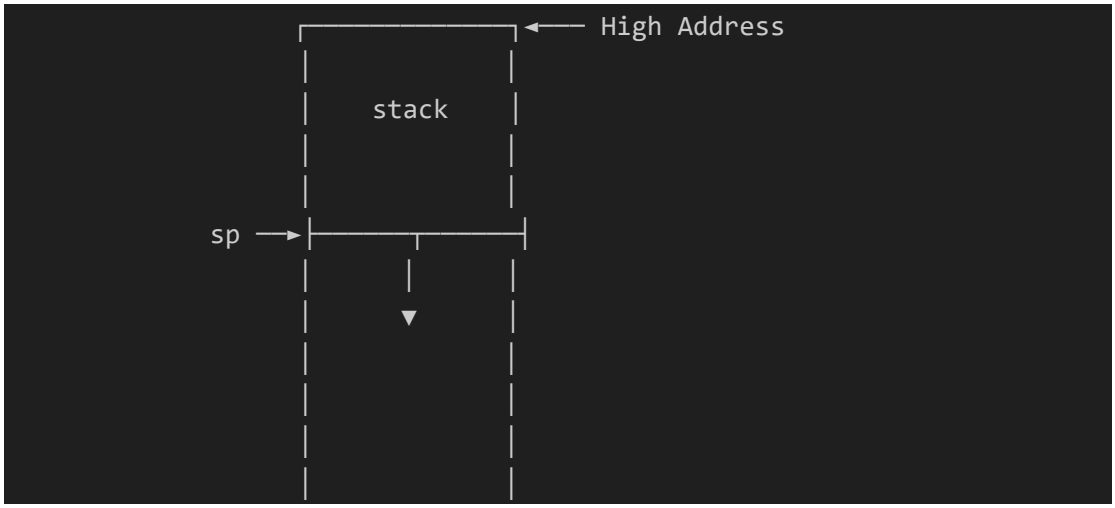
在初始化时需要用一些自定义的宏，因此需要在 defs.h 中添加如下内容：

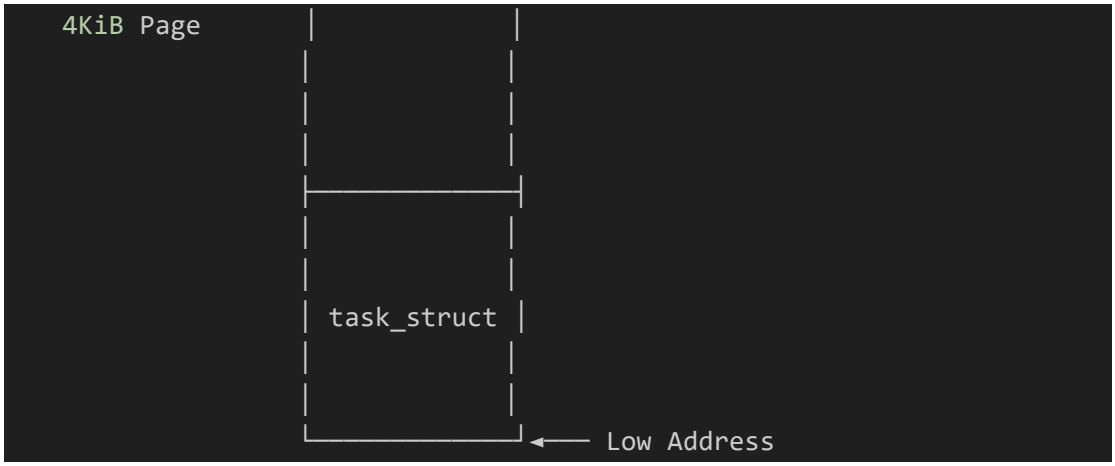
```
#define PHY_START 0x0000000080000000
#define PHY_SIZE 128 * 1024 * 1024 // 128 MiB, QEMU 默认内存大小
#define PHY_END (PHY_START + PHY_SIZE)

#define PGSIZE 0x1000 // 4 KiB
#define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
#define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))
```

（二）线程调度功能实现

1. 线程初始化

在初始化线程的时候，我们参考 Linux v0.11 中的实现为每个线程分配一个 4 KiB 的物理页，我们将 task_struct 存放在该页的低地址部分，将线程的栈指针 sp 指向该页的高地址。具体内存布局如下图所示：





当我们的 OS 运行起来的时候，其本身就是一个线程（idle 线程），但是我们并没有为它设计好 task_struct，所以第一步我们要：

为 idle 设置好 task_struct 的内容；

将 current，task[0] 都指向 idle；

为了方便起见，我们将 task[1] ~ task[NR_TASKS - 1]全部初始化，这里和 idle 设置的区别在于要为这些线程设置 thread_struct 中的 ra 和 sp，代码如下：

```
1.    void task_init() {
2.        srand(2024);
3.
4.        // 1. 调用 kalloc() 为 idle 分配一个物理页
5.        // 2. 设置 state 为 TASK_RUNNING;
6.        // 3. 由于 idle 不参与调度，可以将其 counter / priority 设置为 0
7.        // 4. 设置 idle 的 pid 为 0
8.        // 5. 将 current 和 task[0] 指向 idle
9.
10.       idle = (struct task_struct*) kalloc();
11.       idle->state = TASK_RUNNING;
12.       idle->counter = 0;
13.       idle->priority = 0;
14.       idle->pid = 0;
15.       current = idle;
16.       task[0] = idle;
17.
18.       // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始
    化
19.       // 2. 其中每个线程的 state 为 TASK_RUNNING, 此外,
    counter 和 priority 进行如下赋值:
20.       //     - counter  = 0;
```

```
21.       //      - priority = rand() 产生的随机数（控制范围
   在 [PRIORITY_MIN, PRIORITY_MAX] 之间）
22.       // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中
   的 ra 和 sp
23.       //      - ra 设置为 __dummy（见 4.2.2）的地址
24.       //      - sp 设置为该线程申请的物理页的高地址
25.
26.       for (int i = 1; i < NR_TASKS; i++){
27.           task[i] = (struct task_struct*) kalloc();
28.           task[i]->state = TASK_RUNNING;
29.           task[i]->counter = 0;
30.           task[i]->priority = rand() % (PRIORITY_MAX - PRIORITY_MIN
    + 1) + PRIORITY_MIN;
31.           task[i]->pid = i;
32.           task[i]->thread.ra = (uint64_t) &__dummy;
33.           task[i]->thread.sp = (uint64_t) task[i] + PGSIZE;
34.       }
35.
36.       printk("...task_init done!\n");
37.   }
```

在 arch/riscv/kernel/head.S 中合适位置处调用 task_init() 进行线程初始化：

```
1.    la sp, boot_stack_top #store the address of stack top
2.    jal mm_init
3.    jal task_init
```

2. __dummy 与 dummy 的实现

task[1] ~ task[NR_TASKS - 1]都运行同一段代码 dummy()

当线程在运行时，由于时钟中断的触发，会将当前运行线程的上下文环境保存在栈上；当线程再次被调度时，会将上下文从栈上恢复，但是当我们创建一个新的线程，此时线程的栈为空，当这个线程被调度时，是没有上下文需要被恢复的，所以我们需要为线程第一次调度提供一个特殊的返回函数 __dummy：

在 arch/riscv/kernel/entry.S 中添加函数 __dummy：

在 __dummy 中将 sepc 设置为 dummy() 的地址，并使用 sret 从 S 模式中返回

```
1.        .extern dummy
2.        .global __dummy
3.    __dummy:
4.        la t0, dummy
5.        csrw sepc, t0
```

```
6.          sret
```

3. 实现线程切换

判断下一个执行的线程 next 与当前的线程 current 是否为同一个线程，如果是同一个线程，则无需做任何处理，否则调用 __switch_to 进行线程切换：

```
1.    void switch_to(struct task_struct *next) {
2.        if (next == current){
3.            return;
4.        }else{
5.            struct task_struct *prev = current;
6.            current = next;
7.            __switch_to(prev, next);
8.        }
9.    }
```

在 entry.S 中实现线程上下文切换 __switch_to：

__switch_to 接受两个 task_struct 指针作为参数；

保存当前线程的 ra，sp，s0~s11 到当前线程的 thread_struct 中；

将下一个线程的 thread_struct 中的相关数据载入到 ra，sp，s0~s11 中进行恢复：

```
1.    __switch_to:
2.        addi t0, a0, 32 # save state to prev process
3.        sd ra, 0*8(t0)
4.        sd sp, 1*8(t0)
5.        sd s0, 2*8(t0)
6.        sd s1, 3*8(t0)
7.        sd s2, 4*8(t0)
8.        sd s3, 5*8(t0)
9.        sd s4, 6*8(t0)
10.       sd s5, 7*8(t0)
11.       sd s6, 8*8(t0)
12.       sd s7, 9*8(t0)
13.       sd s8, 10*8(t0)
14.       sd s9, 11*8(t0)
15.       sd s10, 12*8(t0)
16.       sd s11, 13*8(t0)
17.
18.       addi t0, a1, 32 # restore state from next process
19.       ld ra, 0(t0)
20.       ld sp, 8(t0)
21.       ld s0, 2*8(t0)
```

```
22.         ld s1, 3*8(t0)
23.         ld s2, 4*8(t0)
24.         ld s3, 5*8(t0)
25.         ld s4, 6*8(t0)
26.         ld s5, 7*8(t0)
27.         ld s6, 8*8(t0)
28.         ld s7, 9*8(t0)
29.         ld s8, 10*8(t0)
30.         ld s9, 11*8(t0)
31.         ld s10, 12*8(t0)
32.         ld s11, 13*8(t0)
33.
34.         ret
```

4. 实现调度入口函数

实现 do_timer() 函数，并在 trap.c 时钟中断处理函数中调用：

```
1.    void do_timer() {
2.        // 1. 如果当前线程是 idle 线程或当前线程时间片耗尽则直接进行调度
3.        // 2. 否则对当前线程的运行剩余时间减 1，若剩余时间仍然大于 0 则直接返回，否则进行调度
4.
5.        if (current == idle) {
6.            schedule();
7.        } else {
8.            --current->counter;
9.            if ((long)(current->counter) > 0) return;
10.           else {
11.               current->counter = 0;
12.               schedule();
13.           }
14.       }
15.   }
```

5. 线程调度算法实现

本次实验我们需要参考 Linux v0.11 调度算法代码实现一个优先级调度算法，具体逻辑如下：

task_init 的时候随机为各个线程赋予了优先级

调度时选择 counter 最大的线程运行

如果所有线程 counter 都为 0，则令所有线程 counter = priority

即优先级越高，运行的时间越长，且越先运行

设置完后需要重新进行调度

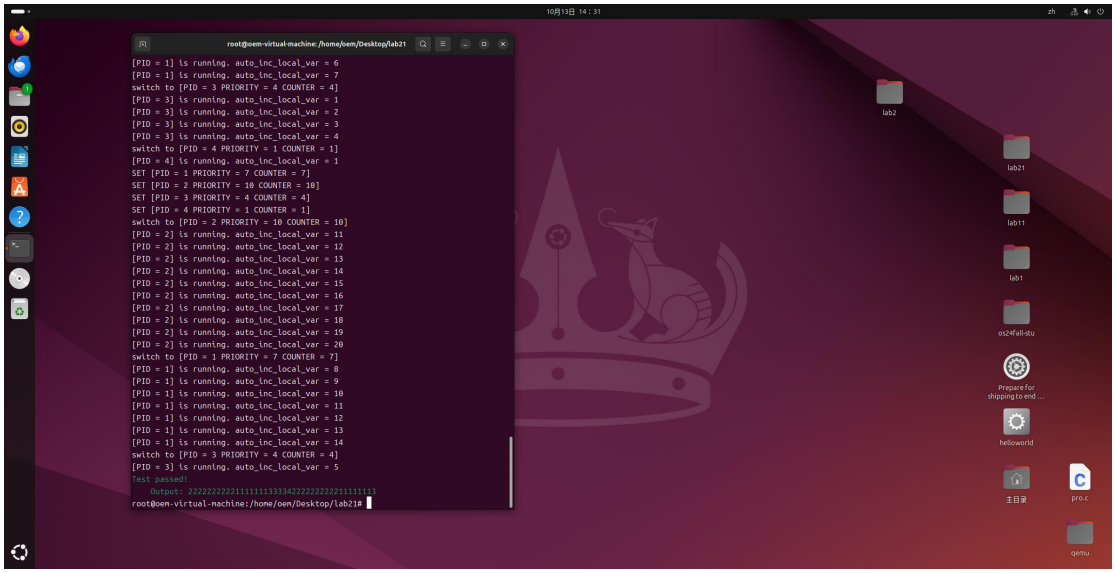
最后通过 switch_to 切换到下一个线程

```
1.    void schedule() {
2.        int i,next,c;
3.        while (1){
4.            c = -1;
5.            next = 0;
6.            i = NR_TASKS;
7.            while (--i){
8.                if (!task[i]){
9.                    continue;
10.               }
11.               if (task[i]->state == TASK_RUNNING && (long)(task[i]-
   >counter) >= c){
12.                   c = task[i]->counter;
13.                   next = i;
14.               }
15.           }
16.           if (c){
17.               printk("switch to [PID = %d PRIORITY = %d COUNTER = %
   d]\n", next, task[next]->priority, task[next]->counter);
18.               break;
19.           }
20.           for (i = 1; i < NR_TASKS; i++){
21.               if (task[i]){
22.                   task[i]->counter = task[i]->priority;
23.                   printk("SET [PID = %d PRIORITY = %d COUNTER = %d]
   \n", i, task[i]->priority, task[i]->counter);
24.               }
25.           }
26.       }
27.       switch_to(task[next]);
28.   }
```

（三）编译及测试

为了验证算法正确性，本次实验加入了一个测试样例（在 4 个线程的情况下的 pid 输出）

测试结果如下：



根据图中显示，测试通过

三、讨论和心得

本次实验建立在 lab1 环境已经搭建好的基础上，做起来并不是特别困难，但是仍然有几个需要注意的点。一般来说，程序在调用完某个函数后，代码上后面的语句会立刻执行，但是这个程序中有个例外，当调用__switch_to()函数后，ra 寄存器读取了其他的值，也就是说执行完该函数后返回的地址被改变了，所以在 c 代码中执行完__switch_to()后面的代码不会被立刻执行，而这也是程序运行线程的逻辑，理解这点对代码的编写很有帮助。

这次实验让我提升了汇编代码的能力以及对 RISC-V 架构下寄存器的理解，同时也让我了解到了线程调度的原理。

四、思考题

1.

C 语言在调用__switch_to()函数的时候，会将 RISV-V 中的通用寄存器中由调用者保存的寄存器压入栈，因此__switch_to()只需要保存 C 语言没有保存的寄存器，即 sp 和 s0~s11 寄存器。而在调用完__switch_to()之后需要返回指定的地址继续运行程序，因此还需要保存 ra 寄存器的值。因此总共要保存 14 个寄存器。

2.

mm_init()函数

释放内存范围：kfreerange 函数被调用，用来释放从 _ekernel（内核结束的地址）到 PHY_END（物理内存结束的地址）之间的内存。这段内存通常是内核加载到内存后的剩余可用物理内存范围。

_ekernel 是内核代码结束的位置，也就是内核占用的最高地址。PHY_END 是系统物理内存的总结束地址。通过 kfreerange，内存管理器将这段物理内存分成多个页（page），并把这些页添加到空闲列表中。

输出日志：初始化完成后，printk 输出一条日志，表明内存管理初始化已完成。

kalloc()函数

功能：kalloc 分配一个内存页。

过程：

从 kmem.freelist（空闲链表）中取出第一个空闲页（struct run 指针 r 指向该页）。

将 kmem.freelist 指向下一个空闲页（r->next）。

调用 memset 将分配的页清零，确保页中的数据被初始化为 0，防止使用未初始化数据带来的问题。

返回该内存页的地址。

kfree()函数

功能：kfree 释放一个内存页，并将其重新加入到空闲链表中。

过程：

地址对齐：因为地址可能没有对齐到页边界，kfree 首先通过位运算将地址对齐到最近的页边界（PGSIZE 的整数倍）。

清零内存：将该页的内容清零，确保数据被完全清除，防止数据泄露或数据污染。

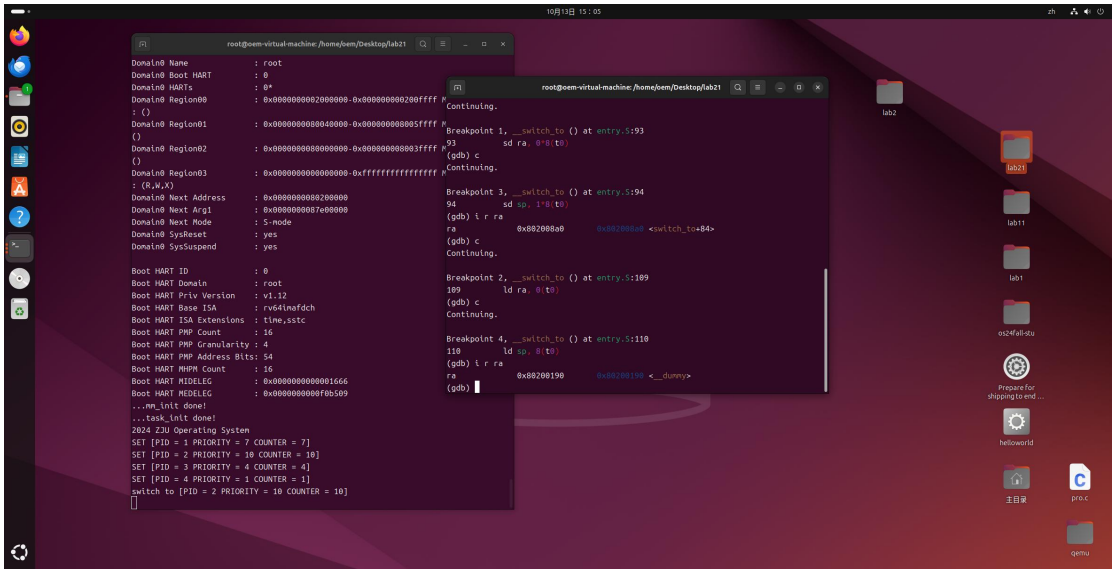
加入空闲链表：将该页的地址转换为 struct run 结构，然后将其插入到空闲链表的头部，维护空闲内存的链表结构。

更新链表头：将空闲链表的头指针更新为刚释放的这个页。

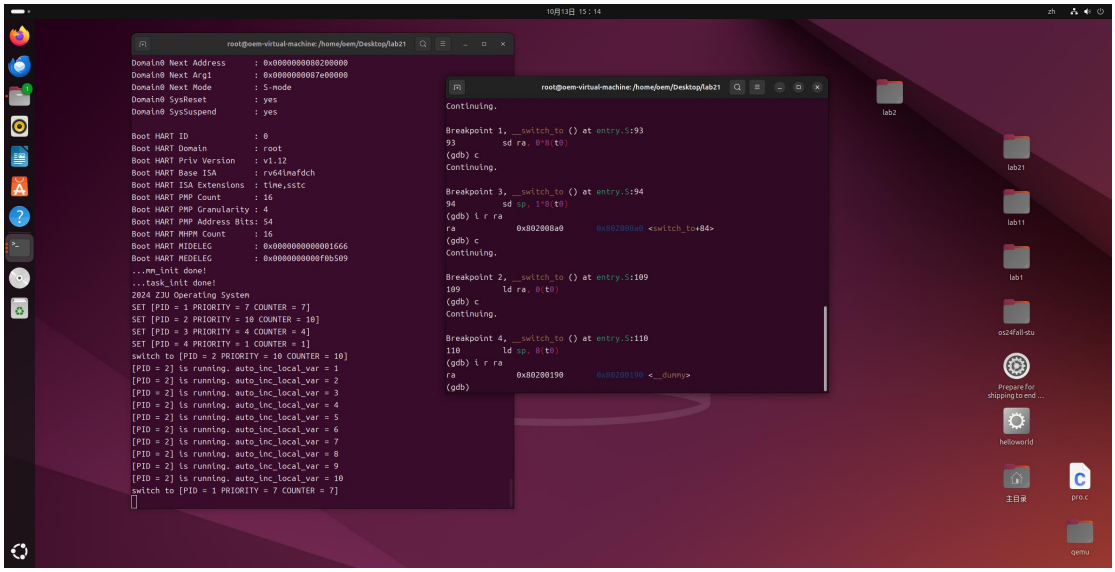
3.

在__switch_to()函数中，保存 ra 的指令在 entry.S 第 93 行，恢复 ra 的指令在 entry.S 第 109 行，因此我们在 entry.S 第 94 行和第 116 行设置断点（即 ra 后的下一条指令），以查看 ra 的值。为了方便调试，我们将 NR_TASKS 的值设置为 4。

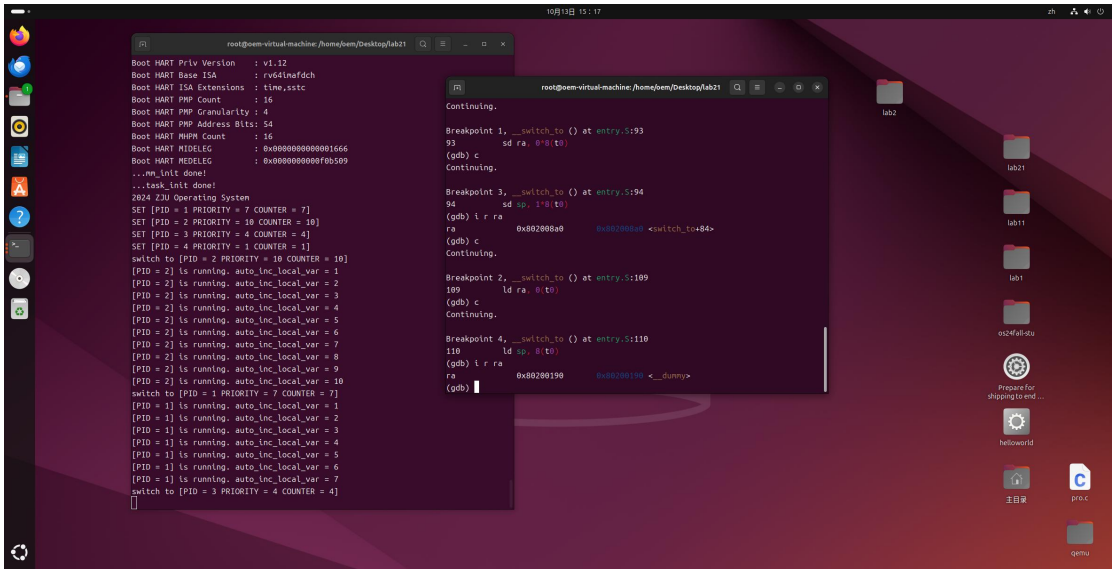
第一次线程调用是从 idle 切换到 task[2]，保存的 idle 的 ra 为<switch_to+84>，恢复的 task[2] 的 ra 为<__dummy>。



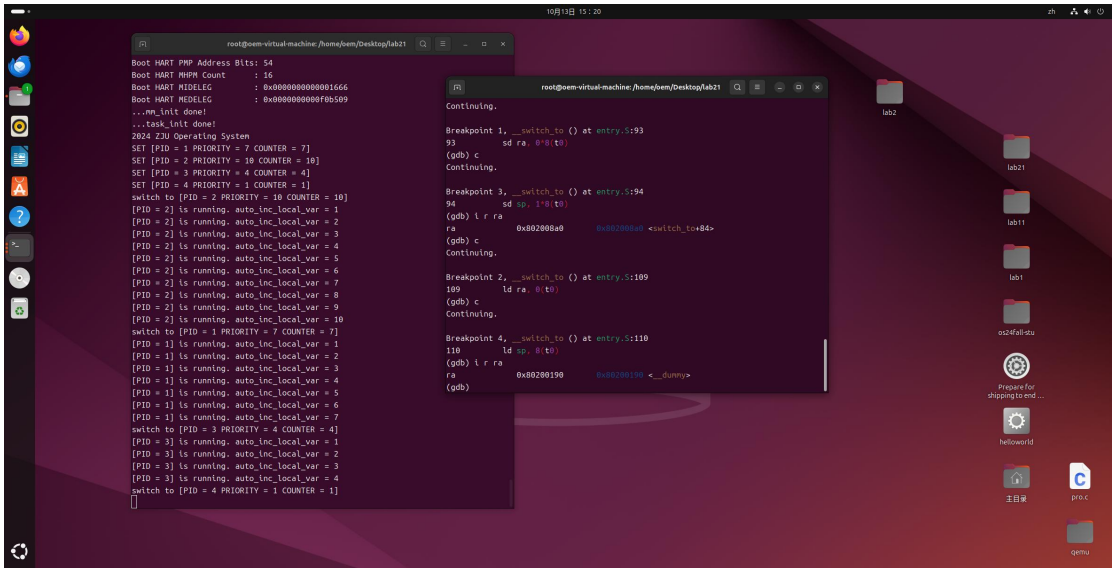
第二次线程调用是从 task[2]切换到 task[1]，保存的 task[2]的 ra 为<switch_to+84>，恢复的 task[1]的 ra 为<__dummy>。



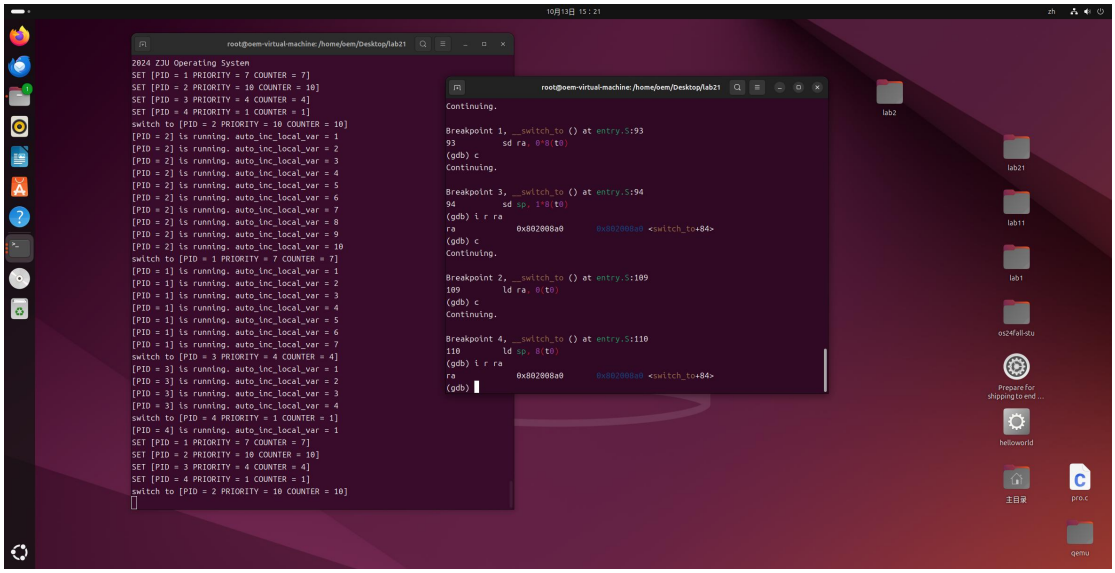
第三次线程调用是从 task[1]切换到 task[3]，保存的 task[1]的 ra 为<switch_to+84>，恢复的 task[3]的 ra 为<__dummy>。



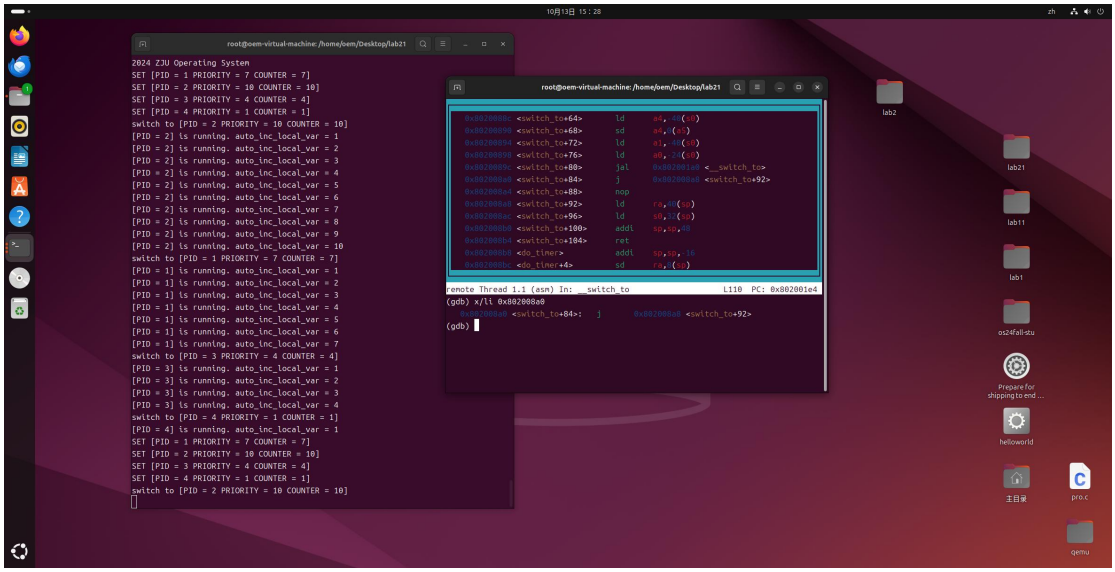
第四次线程调用是从 task[3]切换到 task[4]，保存的 task[3]的 ra 为<switch_to+84>，恢复的 task[4]的 ra 为<__dummy>。



第五次线程调用是从 task[4]切换到 task[2]，保存的 task[4]的 ra 为<switch_to+84>，恢复的 task[2]的 ra 为<switch_to+84>。

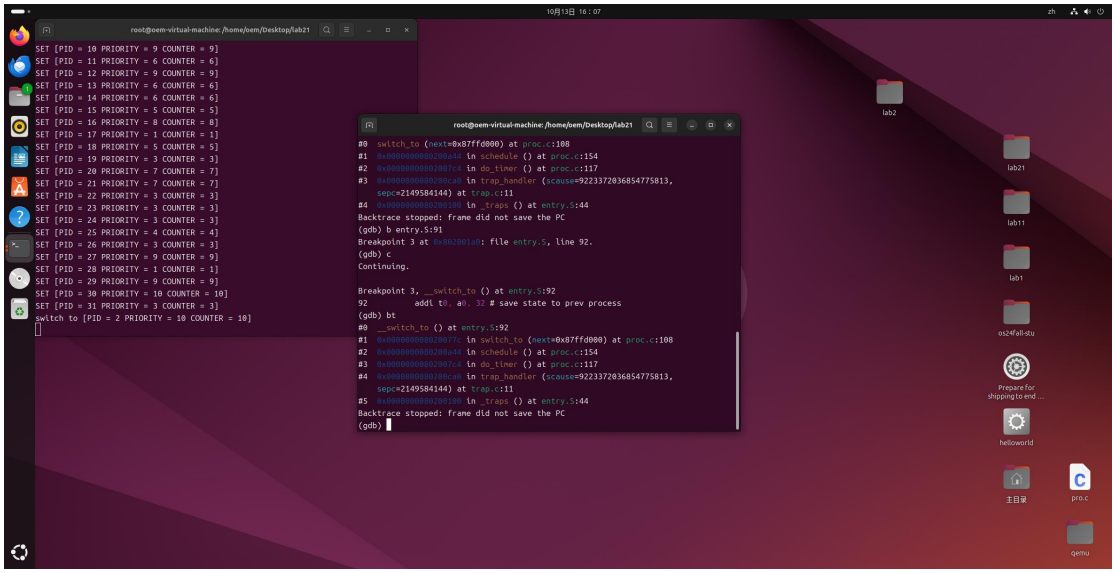


不难推出，接下来保存的 ra 和恢复的 ra 都是<switch_to+84>。
查看该地址的汇编代码，发现这是 switch_to()函数的末尾。



切换线程的过程中，最后一次 C 语言代码中的函数调用是__switch_to()，所以被切换时，保存的 ra 就是__switch_to()指令的下一条地址<switch_to+84>，当这个线程又被调度的时候，恢复的 ra 也是这个值。当线程初次调度时，它没有经过任何函数调用，因此保存的 ra 就是初始化中的<__dummy>了。

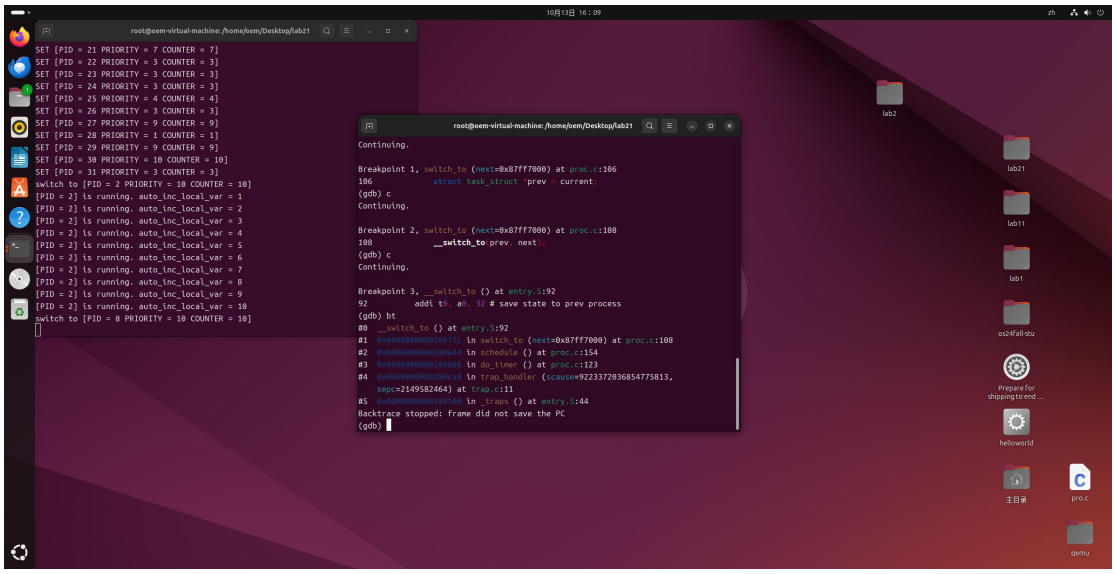
4.
我们在 entry.S 第 92 行设置断点，在运行之后使用 backtrace 查看函数的帧栈布局。
第一次运行后，从 idle 切换到 task[2]。



栈帧 #0：__switch_to
函数: __switch_to
这个函数是内核中上下文切换的底层实现，通常用于保存当前进程的 CPU 上下文并加载下一个进程的上下文。由于它在汇编代码中实现，通常直接与寄存器、栈、线程上下文管理相关。
栈帧 #1：switch_to
函数: switch_to
函数结束地址: 0x000000008020077c
参数: next=0x87ff7000（即下一个要调度的进程的地址或结构体指针）
switch_to 是高层次的上下文切换函数，它调用底层的 __switch_to 来完成真正的上下文切换。它通常会接收两个进程，一个是当前进程，另一个是要切换到的进程。
栈帧 #2：schedule
函数: schedule
函数结束地址: 0x0000000080200a44
schedule 函数是调度器的核心部分，它决定哪个进程要被调度运行。当内核检测到需要进行任务切换时（如时间片用完、进程进入阻塞状态等），它会调用 schedule 函数。
栈帧 #3：do_timer
函数: do_timer
函数结束地址: 0x00000000802007c4
do_timer 通常是时钟中断处理函数的一部分，时钟中断用于驱动调度器。这说明程序执行过程中发生了一个时钟中断，内核响应中断后，检查是否需要进行进程调度。
栈帧 #4：trap_handler
函数: trap_handler
函数结束地址: 0x0000000080200ca0
参数: scause=9223372036854775813，sepc=2149582464
trap_handler 是内核中的陷阱处理器，用于处理异常或中断。在这个栈帧中，scause 和 sepc 分别是陷阱的原因和程序计数器。scause 通常是一个寄存器值，解释了是什么类型的异常或中断触发了陷阱，而 sepc 则指向导致中断的指令地址。
栈帧 #5：_traps

函数: _traps
函数结束地址: 0x0000000080200100
描述: _traps 是陷阱处理的汇编入口，通常用于处理异常、中断或系统调用。在这个栈帧中，异常发生后，控制权从用户态或其他地方转移到这个入口点。

第二次运行后，从 task[2]切换到 task[8]。



栈帧 #0：__switch_to
函数: __switch_to
栈帧 #1：switch_to
函数: switch_to
函数结束地址: 0x000000008020077c
参数: next=0x87ff7000（即下一个要调度的进程的地址或结构体指针）
栈帧 #2：schedule
函数: schedule
函数结束地址: 0x0000000080200a44
栈帧 #3：do_timer
函数: do_timer
函数结束地址: 0x0000000080200808
栈帧 #4：trap_handler
函数: trap_handler
函数结束地址: 0x0000000080200ca0
参数: scause=9223372036854775813，sepc=2149582464
栈帧 #5：_traps
函数: _traps
函数结束地址: 0x0000000080200100

五、附录

无